CS Study Group FY07 Phase 2

Machine-Checked Metatheory for Security-Oriented Languages

Final Technical Report

PIs:	Stephanie Weirich*
	Steve Zdancewic
Mail:	Computer and Information Science Department
	University of Pennsylvania
	3300 Walnut St.
	Philadelphia, PA 19104
Email:	sweirich@cis.upenn.edu
Phone:	215-573-2821
Fax:	215-898-0587

1 Project Introduction

For the past few years, the Department of Defense has been transforming its culture to be based on information sharing. Organizations within the DoD recognize that this change leads to greater effectiveness. For example, the doctrine of network-centric operations emphasizes shared situational awareness for warfighters to enable collaboration and self-synchronization, and to enhance sustainability and speed of command [19]. Combatant Commands use Common Operating Pictures (COPs) that integrate data from a variety of sources, geospatially locating them on a single map. After 9/11, the restructuring of the Intelligence Community (IC) seeks to remove the barriers to joint intelligence work. From the 9/11 Commission Report [53]:

> The importance of integrated, all-source analysis cannot be overstated. Without it, it is not possible to "connect the dots." No one component holds all the relevant information.

However, information sharing is at odds with information security. The primary mechanism that the DoD uses to ensure the secrecy of sensitive information is physical isolation. The NIPRNet (Unclassified but Sensitive Internet Protocol Router Network), SIPRNet (Secret Internet Protocol Router Network) and JWICS (Joint Worldwide Intelligence Communications System) networks compartmentalize unclassified, secret and top secret data. Such airgapping mitigates the potential damage of successful attacks, but it also comes with a cost to effectiveness that is recognized by members of the DoD. When asked to suggest problems for the Computer Science Study Group to consider, the last item on the list that Mr. Richard Corson (USSOCOM JOC Chief) circulated read:

> Cross Domain Capability: Good NSA approved cross domain capability to allow working between UNCLAS, SECRET, and TOP SECRET networks. [1]

During CSSP visits, similar organizations also expressed dissatisfaction with the existing compartmentalization strategy, citing issues such as the cost of multiple wires and machines, the complexity of approved information transfers (such as with USB storage devices), the number of different security levels present when conducting coalition warfare, and the inability to run analysis tools at multiple security levels. As a result, organizations such as USSOCOM, the NSA, and the DIA are actively seeking alternative methods of Information Assurance (IA).

Although any solution to this secure data integration problem will require the synthesis of techniques from the networking, cryptography, and operating systems domains, one crucial component is the security of the user applications themselves. The software should permit authorized users to manipulate and process confidential data from multiple domains, yet it must not allow accidental or malicious violation of the end-to-end security policies in force. Establishing the security of application software is critical because existing OS-level access-control policies are too coarse grained to account for important information flows, such as how parts of a document are manipulated via user edits or transmitted over the network. Moreover, to warrant the NSA's approval, such software must be certified to ensure that it does not contain Trojan horses, covert channels, or other unauthorized code.

Under current best practices, the detailed software auditing needed to both rule out malicious code and verify that the software follows the appropriate security policies is extremely time consuming and expensive. Recent advances in programming language research suggests a better path: develop abstractions that allow programmers to directly express security concerns in the

[1] Richard Corson, "CS2G Projects for the USSOCOM JOC", 23 Aug 2006

program itself. Such *security-oriented languages*, described in more detail below, make it easier and cheaper to develop software that satisfies strong security policies. Of course, to know whether a program written using a security-oriented language meets its IA requirements, it is necessary to understand what properties the programming language itself provides. This so-called *metatheory* problem—the problem of rigorously establishing the properties of a security-oriented programming language—is the main research thrust of this project.

The novel contributions of the project include:

- Basic research that extends the current capabilities of using machine-checked proofs for verifying security properties and metatheory of complex, security-oriented languages.
- The development of a suite of tools and associated methodologies that demonstrate how to apply the basic research, implemented in the context of the Coq proof assistant.
- Machine-checked proofs of *type safety* and *noninterference*, two critical security properties, for a test-bed programming language called AURA, a security-oriented language that will target the Microsoft .NET framework.

Outline The rest of this final report is laid out as follows: The next section describes security-oriented languages in more detail and explains why establishing their *metatheoretic properties* is a critical component in implementing cross-domain secure systems. Along the way, Section 2 also describes the need for machine-checked metatheory for security-oriented languages. Section 3 provides background for AURA, the specific programming language that is the test-bed for this research. The next three sections lay out results of the project: detailing the specific methodology undertaken for the AURA test-bed (Section 4), the design of additional tools that aid in the metatheory of programming languages (Section 5), and the evaluation of our tools and methodology that we performed to ensure that our efforts will apply beyond the verification of the AURA language (Section 6).

2 Security-oriented languages: background and motivation

As observed above, manual auditing of software to determine whether it is secure enough for use in mission-critical and cross-domain contexts is expensive and time consuming. Software verification, once thought to be a panacea for such problems, has not panned out in practice, largely because verifying even relatively small programs takes extraordinary skill and a huge amount of effort. Such an approach is neither cost effective nor scalable.

The story is not entirely bleak, however. Advances in programming language design have created remarkably effective tools that can rule out large classes of software safety problems. For example, modern programming languages like Java and C#, if properly implemented, prevent buffer overflows and a host of similar memory errors that plague C and C++ software.

In practice, the only scalable way to verify properties of large or numerous programs is to provide support during the application development process. The key idea from the programming languages research domain is the use of static type systems that enable the compiler to perform light-weight verification of fundamental safety properties. The win compared to traditional software verification is clear: rather than meticulously verifying each program individually, a type system guarantees properties about *all* programs accepted by the compiler of a safe language; furthermore the analysis can be mostly automated, permitting it to scale to large software. There

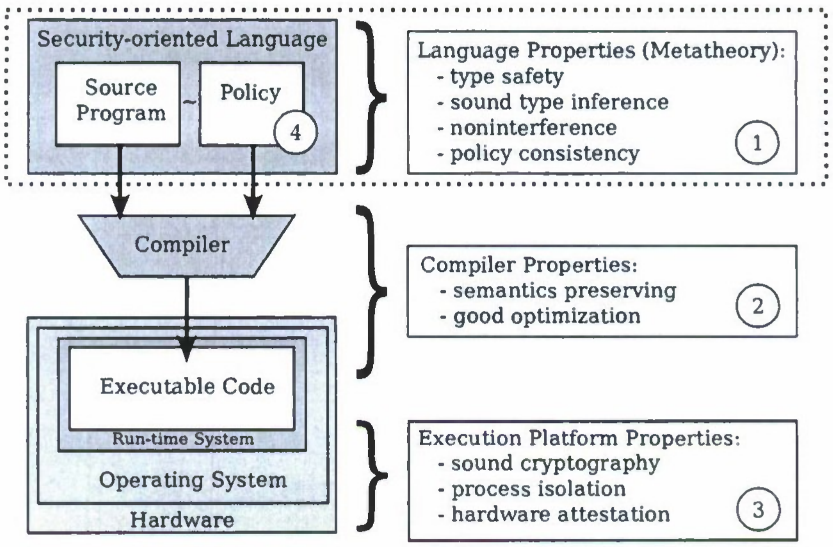

Figure 1: Architecture of a system built using a security-oriented language. On the left are the main hardware and software artifacts that go into the development and deployment of a networked information system. On the right are three key requirements for establishing the security of the system; the correctness of the policy is a fourth criterion. This proposal's primary research thrust is machine-checked metatheory: using computer-assisted proof development to validate the components in the dotted box.

is still an obligation to verify that the language's type system ensures the property of interest—that is, to establish the language's *metatheory*—but this task can be undertaken once and for all by the expert programming language designers and implementors. Ordinary programmers get the safety benefits, without having to prove their programs correct by hand.

The advent of the Internet and the rise of security concerns have spawned a vibrant research community focused on *security-oriented programming languages* [56, 1, 38, 44, 6, 52, 21], of which perhaps the best known is Jif, developed by Myers *et al.* at Cornell [39]. Security-oriented languages aim to achieve the scalability benefits of type systems for security properties beyond traditional memory safety. Rooted in the pioneering work by Denning in the 1970s [16, 17], the goal is to provide strong enforcement of information-flow and access-control policies, language support for authorization, authentication and audit operations, and seamless integration with cryptography[2]. Together, these features combine to provide a software-development platform that should significantly ease the burden of building certifiably secure programs suitable for cross-domain and mission critical applications. The need for such language-based security has been articulated both by studies commissioned by the U.S. Government [47] and by leaders in the academic community [48].

Figure 1 shows the key components of a system built using a security-oriented programming language. As indicated at the top left of the figure, the programmer writes the software using familiar datatypes and algorithms, but in addition to these usual abstractions, the language itself provides mechanisms for describing confidentiality and integrity policies, authorization requirements, access control policies, and the like. Importantly, this design allows the information assurance policies to be expressed in terms of concepts that make sense for the application at hand (such as 'document' or 'e-mail') , rather than the abstractions (such as 'file' or 'machine word')

[2]For an overview of just some of the work in this area, see Sabelfeld and Myers survey [46]

provided by the operating system.

Given the policy information and the program, the language implementation can verify, by type checking, that the program meets the requirements of the policy—if the program does not meet the requirements, it is rejected as "bad" and not accepted. For security-oriented languages that enforce information-flow policies, there are two most relevant properties. The first is *type safety*, which demonstrates that the language's abstractions are not violated (ruling out many safety errors) and provides a solid foundation on which to build higher-level security properties. The second is *noninterference* [22], which ensures that the only information flows through the program are permitted by policy, thereby ensuring that confidential information is not inadvertently (or maliciously) leaked to publicly accessible parts of the system.

Correctness of the secure system Returning to Figure 1, there are two more pieces of the diagram that play a rôle in the security of the system. First, the language compiler, in addition to producing bytecode or machine instructions that can be directly executed, is also responsible for ensuring that the security policy is respected by the resulting executable code. Second, when the program is run it will typically rely on features (such as process isolation) and services (such as cryptography, the file system, or network access) provided by the underlying operating system and hardware.

The security of the system therefore depends on at the correctness of at least four things marked in the Figure: (1) the properties guaranteed by the type system, namely that the program meets its policy, (2) the program transformations and optimizations done by the compiler, (3) the network protocols, cryptography, etc. implemented in the hardware and OS, and, (4) the security policy itself. Not surprisingly, because finding solutions for each of these correctness problems is vital to creating secure software systems, all four have received considerable attention from academia, the government, and industry.

Until recently, the best practice for establishing these correctness criteria was painstaking manual audit, code reviews, and pencil and paper proofs checked by experts. Now the trend is to use computer-aided verification techniques such as model checking, theorem provers, and mechanized proof assistants, which enable rigorous auditing to scale to larger, more complex systems than possible by hand. To achieve the high-confidence standards required for the cross-domain and network-centric information applications envisioned by agencies in the DoD, it will be necessary to apply such computer aided verification techniques to the four criteria mentioned above.

To date, there has been significant progress made in using these tools to address the correctness criteria outlined above. A number of projects address the problem of compiler correctness (criterion (2)), including: Leroy's machine-checked compiler back end for C [35]; Appel *et al.*'s Foundational Proof Carrying Code project [37, 3, 2], and Crary's formalization of Foundational Typed Assembly Language (TALT) [15]. With respect to trusted run-time systems (criterion (3)) there have also been a variety of relevant projects: Sewell *et al.*'s machine-checked specification of the TCP/IP protocol stack [11], Shi's [50] and Zhou *et al.*'s [57] work on verified device drivers, and the Microsoft Singularity operating system [26] among others. There are also a variety of tools for policy analysis, mostly aimed at determining consistency properties for access-control (criterion (4)) [24, 18, 9, 36].

For criterion (1), however, there has been relatively little work: Klein and Nipkow have a machine-checked model of a Java-like language [33] and Harper *et al.* have verified the type system of Standard ML of New Jersey [34]. Both of these projects address only type safety—the programming languages they consider aren't security-oriented and therefore lack support for strong information-flow or access control security policies. To our knowledge there are few instances of machine-checked metatheory for security-oriented languages. Most were undertaken for small, toy programming languages: David Naumann verified a secure information flow analyzer for a fragment of the Java language [40], and Jacobs, Pieters and Warnier [27] showed noninterference for a simple imperative language in the PVS theorem prover. Also, Strecker showed noninterference for MicroJava in Isabelle/HOL [51]. Barthe *et al*'s work on a secure variant of Java bytecode [8] is the most comprehensive, but is aimed at bytecode, not source-level security properties.

2.1 The missing piece: Mechanized metatheory for security-oriented languages

The primary goal of this project was to address criterion (1) head-on: *We propose to conduct basic research about how to produce machine-checked proofs of the metatheory for security-oriented languages.*

This is a worthwhile and significant task for several reasons. As we have seen, the security of programs written in a security-oriented language depends crucially on the correctness of its metatheory. Just as an unsound type system may create an opportunity for an attacker to exploit a buffer overflow, any flaw in the noninterference analysis of a security-oriented language might permit an application to leak sensitive information. This problem is particularly alarming in the cross-domain network operations setting that is the primary interest of the organizations within the DoD.

This task is also substantial and challenging. Security-oriented languages are complex artifacts. They offer a strong safety net to application programmers, but the security properties such as safety and noninterference are difficult for the implementors to establish rigorously. Unfortunately, rigorous proofs even about simpler programming languages are difficult for humans to manage: they are long and tedious, with just a few interesting cases sprinkled among boring ones (which must nevertheless be checked carefully to confirm that they really *are* boring!). The difficulty of these proofs arises from the management of many details rather than from deep conceptual difficulties (these arise earlier, in the process of getting the definitions right); yet small mistakes or overlooked cases can invalidate large amounts of work. These effects are naturally amplified as languages scale.[3]

Automated proof assistants offer the hope of significantly easing these problems. However, despite encouraging progress in this area in recent years and the availability of several mature tools (including Coq [10], Isabelle [41], ACL2 [32], and Twelf [43]), the application of these tools to programming language problems is not commonplace.

Concretely, we will address these issues:

- **Security-property specific techniques:** Security-oriented languages require domain-specific proof techniques to establish type safety and noninterference (the fundamental property for information-flow assurance). Our hypothesis is that addressing this problem will require a non-standard operational semantics and additional library support.
- **Proof overhead and management:** Managing the details of large, complex proofs is quite difficult. Furthermore, those proofs must be robust with respect to the changes of an evolv-

[3]A typical example of the current state of affairs of programming language metatheory can be found in a recent conference paper by Chen and Tarditi [13] on a new typed intermediate language for compiling object-oriented source languages such as C#. The safety theorem for this language is stated in a standard form ("execution of a well-typed program cannot get the abstract machine into an erroneous state"), and the proof of this fact is regarded by the authors as too boring even to sketch in detail: the paper just says "By standard induction over the typing rules." However, the details of this "standard" proof, in the accompanying technical report, run to 26 dense pages! What, realistically, are the odds that anyone besides the authors has checked it carefully? Another point worth noting is that the paper's second author is one of the lead compiler developers at Microsoft: this is not just an academic exercise!

ing language design. We have some experience solving this problem for specific tasks (such as how to handle variable binding [4]), but we need more automation.

- **Generality:** Our efforts will have more impact if our results are general and applicable to the specifications of other security-oriented languages. To ensure the robustness of our methodology, we will encapsulate the reusable parts of our infrastructure into libraries, use those to formalize a middleweight version of Jif, and analyze our own design evolution.

In addition to the basic research needed to address these challenges, the output of this project will be twofold: First, we plan to mechanize the metatheory of a test-bed security-oriented language called AURA that will serve as a specific test case of the problems mentioned above. The development of the AURA language itself is of independent merit, since it represents a cutting-edge security-typed language. Second we will develop new tools and processes so that the metatheory of similar languages may also be more easily mechanically verified, thereby making our results accessible to other researchers and practitioners in the field.

Before explaining our approach to solving these challenges and our expected contributions in more detail, it is first necessary to describe AURA, the security-oriented language that will serve as the testing ground for our tools and techniques.

3 AURA: a security-oriented language test bed

AURA is a new *security-oriented language* developed at the University of Pennsylvania. AURA's programming model smoothly integrates information-flow and access control constraints with the cryptographic enforcement mechanisms necessary in a distributed computing environment, making it ideal for the kinds of cross-domain applications wanted by organizations within the DoD. The design of the AURA language is summarized in the paper "AURA: A Programming Language for Authorization and Audit", presented at the International Conference on Functional Programming, September 2008 [28] and extended in Jeffrey Vaughan's dissertation [54], and the paper ""Encoding Information Flow in AURA" [29].

The key innovation in the AURA project is the pervasive use of *policy-justified data*, a technique that simultaneously generalizes both traditional multi-level security (MLS) labels [16, 20, 22, 46] and capability-based trust-management mechanisms [12, 31]. Like MLS, policy-justified data associates security-policy-specific metadata with each data value in the system and propagates them together to account for information flows. As in capability-based trust-management systems, confidentiality and integrity policies for information-flow, access-control, and downgrading are expressed using a rich policy logic that can be implemented using standard cryptographic techniques (e.g., digital signatures). Unlike either of these techniques in isolation, however, their combination in AURA provides remarkable synergy: Incorporating trust-management-style authorization logics into the programming language allows the AURA compiler to ensure that programs do appropriate access-control checks and logging, while using an authorization logic to describe the security-levels of data in the system permits flexible, decentralized, and dynamic enforcement of information-flow policies.

AURA provides following features:

- A decentralized authorization logic that enables software developers and system administrators to uniformly and declaratively specify access-control and information-flow policies, including those that include downgrading (declassification and endorsement). The policies are intended to support both confidentiality and integrity policies.

- Support for authentication using a first class notion of *principal* that gives the policy language a natural interpretation using public-key infrastructure (PKI).
- Support for policy-justified data that connects the data being manipulated by the program with run-time capabilities that carry both MLS policy and audit information.
- Language support for auditing that ensures that appropriate log entries are generated whenever a security-relevant operation is performed by the system. This audit trail includes the evidence that justifies the action, not just a record of the action itself.

One important facet of the AURA language is that it requires system programmers to generate informative logs that explain *why* the information being processed has been released to various constituents—logging is *mandatory*, not *discretionary*. This mandatory audit-trail is crucial for helping find flaws both in the software components and in the policy components.

The AURA project implementation consists of a compiler and language run-time system that provide appropriate compliance checking of authorization certificates, suitable cryptographic algorithms, and efficient means of logging and manipulating the policy metadata. AURA is implemented using Microsoft's .NET framework (specifically, the compiler will be written in the F# language). Furthermore, the AURA compiler targets the .NET framework. This strategy provides two important benefits: First, the .NET platform offers a path to interoperability with a wide array of existing code and libraries. Second, the .NET framework provides existing support that simplifies the implementation of compilers—our AURA compiler can take advantage of work already done by the Microsoft developers.

One could implement a secure system in any programming language, but using a security-oriented language like AURA provides a number of significant advantages: The compiler can statically detect (via type checking) many illegal information flows through the application code, thus yielding a high-degree of confidence in the security of the application itself. AURA can also ensure that the program adheres to the specified access-control policy in force and, assuming that the library interfaces have been implemented correctly (i.e., they are part of the trusted computing base), AURA can ensure the appropriate logging of all information needed to justify the application's behavior.

The AURA language implementation is available for download from `http://www.cis.upenn.edu/~stevez/sol/aura.html`.

4 Mechanizing the Metatheory of AURA

To explain the correctness of AURA's policy enforcement mechanisms, it is necessary to give a coherent theoretical account of the combination of the features mentioned above. Beyond establishing that the programs verified by AURA's type checker meet their intended security policies, such theoretical foundations are necessary to justify various optimizations for compressing audit log information. The theoretical foundations are also crucial for building tools that help administrators understand the impact of changes to a system's authorization policy.

Therefore, a concrete output of this project is the mechanically certified proofs of the metatheoretic properties of the AURA programming language. By verifying the security properties of its type system, we obtain a corresponding theorem about every type-correct AURA program "for free." This represents a significant savings over approaches based on verifying the security properties of programs written in languages that provide no such guarantees.

Our research will focus on mechanically verifying that the following important properties hold for all type-correct AURA programs:

Type safety states that such programs do not "get stuck"—i.e., that linguistic abstractions are not violated: memory references access only well-defined, in-scope objects and control-flow only transfers to approved locations. Type safety provides a base level of system security, for example eliminating attacks based on buffer-overruns, pointer forging, stack smashing, etc.

Noninterference states that such programs regulate the flow of information by their security policies. Public values and outputs are unaffected by sensitive data, except by carefully annotated declassification points. Noninterference is the cornerstone of language-based information assurance.

To reduce complexity, AURA is specified by an *elaboration semantics* [25]. This process defines AURA in terms of a simpler internal language, called Core-AURA. The translation between these two languages is called *elaboration*. The elaboration process eliminates shallow syntactic differences between similar constructs and shrinks the number of features in the core, effectively modularizing the overall semantics. Elaboration also includes analysis and type inference, allowing programmers to elide type and security annotations that may be deduced from context.

The external AURA language makes life easier for programmers, because they have a large grammar of pithy linguistic constructs to draw from, and because they must spend less time writing "obvious" facts about their programs. On the other hand, the overall size of AURA and the elision of these annotations makes it harder to reason about than Core-AURA. It is much simpler to prove properties such as type safety and noninterference for Core-AURA than for AURA.

By formalizing the property of *elaboration soundness*, which states that programs that type check in AURA also type check in Core-AURA, we can lift properties shown for the core language where their proofs are tractable, to the source language, where programming is convenient. Therefore we can divide the metatheoretic task of AURA into three pieces: showing type safety and noninterference for Core-AURA, and showing elaboration soundness.

We used the Coq proof assistant [10] for this task. The Coq tool has already demonstrated its usefulness for formalized metatheoretic reasoning [35, 8, 14, 4]. Coq is not a theorem prover; the normal mode of operation is to check proofs expressed in its logic, the Calculus of Inductive Constructions. To facilitate proof creation, Coq supports interactive proof development through an extensible scripting language. The Coq proofs of type soundness and elaboration soundness for AURA are available from `http://www.cis.upenn.edu/~stevez/sol/aura.html`.

4.1 Noninterference

In the paper "Encoding Information Flow in AURA" [29] we used ideas inspired by a Haskell library for light-weight information-flow security [45] to encode information-flow types in AURA. Our advantage over the Haskell approach is that we can use constructs for AURA s authorization logic for the encoding. The main idea of our encoding is that we use principals to represent security labels, and the type for a secret of type t protected at level H can be encoded as $(x :$ pf H says $Reveal) \rightarrow t$. Intuitively, without Hs private key, no one can create an assertion of the type H says $Reveal$ and therefore secrets protected at level H can not flow to public channels. The noninterference theorem of such encoding depends upon the noninterference properties of the authorization logic. Furthermore, expressive access-control policies specified in authorization logic can be used to specify the policies for declassification.

This work makes the following contributions:

- We show how to encode information-flow types using authorization logics based on prior work [55, 28].
- We prove the basic noninterference theorem of our encoding. The key components of the proof are mechanized in the proof assistant Coq.
- We investigate through examples how declassification can be governed by access-control policies.

4.2 Confidentiality Extension

Jeffrey Vaughan extended AURA with support for creating, manipulating and accessing confidential data [54]. In the $\text{AURA}_{\text{conf}}$ extension, confidential data and computations are given special types and are automatically encrypted as needed. For example, the type **int for Alice** represents an integer readable only by principal Alice. In this system, any user can build secret computations for any other. To unprivileged users, confidential data values and computations are opaque. Values with **for**-types are encrypted, providing security against adversaries that are outside the system and able to, for example, intercept network traffic.

$\text{AURA}_{\text{conf}}$ integrates a novel mix of conventional and new ideas to provide intuitive confidentiality operators. The language contains ciphertexts as first-class values. To enable precise, typed-based analysis of these entities, the typechecker can access statically available private keys and examine ciphertexts at compile time. When an appropriate key cannot be found, facts about particular ciphertexts may be used by the type-checker.

In addition to syntactic soundness, $\text{AURA}_{\text{conf}}$ satisfies a noninterference property that gives one precise, but narrow, characterization of the languages security benefits. Vaughan's work on the $\text{AURA}_{\text{conf}}$ metatheory included a mechanized proofs of both of these properties. These proofs were based on the mechanical proofs of the same properties of Core-AURA, demonstrating that such proofs are amenable to reuse and extension.

5 Eliminating Boilerplate

The most promising way to prove metatheoretic results such as *type soundness* and *noninterference* for realistic programming languages is to do so with mechanical assistance. In particular, Barthe *et al.* report about their experience showing noninterference results for the Java Virtual Machine (JVM) language [7]:

> [W]e have machine-checked our results in the proof assistant Coq in order to gain increased insurance in the soundness proof. Indeed, we feel it is important to resort to proof assistants for managing the complexity of the definitions and proofs involved in establishing noninterference, in particular because the definition of the type system is intricate and the soundness proof involves some lengthy and error-prone proofs by case analysis, as well as some unusual induction principle on the execution of programs.

However, little has been written about the engineering principles that permit such reasoning. Few researchers describe exactly how the proof assistant provides the necessary automation for

```
metavar expvar, x, y, z ::= {{ repr-locally-nameless }}
grammar
exp, e, f, g ::  ::=
 | x       :: :: var
 | e1 e2   :: :: app
 | \ x . e  :: :: abs
   (+ bind x in e +)
substitutions
  single e x :: subst
freevars
  e x :: fv
```

Figure 2: LNgen input file

managing the complexity and sheer size of proofs. Furthermore, our situation differs from previous work in mechanical metatheory verification. Barthe *et al.* started with an existing language. Here, we would like to make mechanical verification a significant part of the language design process.

This co-evolution has its trade-offs. The advantage of formalizing a new language is that we are free to inform the specification of the language based on our experiences with the metatheoretic reasoning. Importantly, it ensures that we are basing our design on sound reasoning principles, and do not commit to inherently untenable design choices.

Furthermore, the fact that verification is part of the design process means that we may define the semantics of the language in such a way so that our proofs are shorter, simpler, and more modular. For example, binding constructs are notoriously difficult to reason about formally—by representing all binding constructs with a single form we can localize that reasoning to one part of the language.

The disadvantage to this process is that as the language grows and evolves, so must its mechanized proof. Therefore, we must pay close attention to the structure of our metatheory so that they do not hinder language evolution. Ideally, we would like the structure to be agile and robust: tweaking the design of the language should result in a manageable amount of change to existing proof text.

5.1 The LNgen tool

To assist with the mechanical formalization of programming language, we developed the LNgen tool. This tool specifically addresses the boilerplate from our methodology for reasoning about programming language metatheory [4]. While we have libraries that factor out reasoning common to all languages, the particular manner in which we represent syntax leads to a proliferation of language-specific lemmas about low-level operations. Their statements and proofs follow directly from the syntax of the language and thus are uninteresting artifacts of our methodology.

The paper "LNgen: Tool Support for Locally Nameless Representations" [5] describes our tool in detail. LNgen uses the same input language as Ott [49], a tool for translating language specifications written in an intuitive syntax into output for LATEX and proof assistants. While Ott generates locally nameless definitionsdatatypes for syntax and relations, functions to calculate free variables and substitutionsfrom the specification, LNgen provides recursion schemes for defining

1. fv-open-upper: fv $(\texttt{open}\ e_1\ e_2) \subseteq$ fv $e_1 \cup$ fv e_2
2. fv-open-lower: fv $e_2 \subseteq$ fv $(\texttt{open}\ e_1\ e_2)$
3. fv-close: fv $(\texttt{close}\ x\ e) =$ fv $e\ x$
4. fv-subst-upper: fv $(\text{subst}\ e_1\ x\ e_2) \subseteq$ fv $e_1 \cup$ (fv $e_2\ x)$
5. fv-subst-lower:(fv $e_2\ x) \subseteq$ fv $(\text{subst}\ e_1\ x\ e_2)$
6. fv-subst-fresh: fv $(\text{subst}\ e_1\ x\ e_2) =$ fv e_2 when $x \notin$ fv e_2
7. subst-fresh-eq: subst $e_1\ x\ e_2 = e_2$ when $x \notin$ fv e_2
8. subst-subst: subst $e_1 x(\text{subst}\ e_2\ y\ e)\ =\ \text{subst}\ (\text{subst}\ e_1\ x\ e_2)\ y\ (\text{subst}\ e_1\ x\ e)$ when $y \notin$ fv e_1 and $y \neq x$
9. subst-open-var: subst $e_1\ x\ (\texttt{open}\ (\texttt{var\_f}y)e_2)\ =\ \texttt{open}\ (\texttt{var\_f}\ y)(\text{subst}\ e_1\ x\ e_2)$ when $x \neq= y$ and lc e_1
10. subst-abs: subst $e_1\ x\ (\texttt{abs}\ e_2) = \texttt{abs}\ (\texttt{close}\ z(\text{subst}\ e_1 x(\texttt{open}\ (\texttt{var\_f}\ z)e_2)))$ when $z \notin$ fv $e_1 \cup$ fv $e_2 \cup x$ and lc e_1
11. subst-close: subst $e_1 x(\texttt{close}\ y e_2) = \texttt{close}\ y(\text{subst}\ e_1 x e_2)$ when $x \neq y$ and $y \notin$ fv e_1 and lc e_1
12. subst-intro: $\texttt{open}\ e_1 e_2 = \text{subst}\ e_1 x(\texttt{open}\ (\texttt{var\_f}x)e_2)$ when $x \notin$ fv e_2
13. open-close: $\texttt{open}\ (\texttt{var\_f}x)(\texttt{close}\ x\ e) = e$
14. close-open: $\texttt{close}\ x(\texttt{open}\ (\texttt{var\_f}x)e) = e$ when $x \notin$ fv e
15. open-inj: $\texttt{open}\ (\texttt{var\_f}\ x)e_1 = \texttt{open}\ (\texttt{var\_f}\ x)e_2$ implies $e_1 = e_2$ when $x \notin$ fv $e_1 \cup$ fv e_2
16. close-inj: $\texttt{close}\ x\ e_1 = \texttt{close}\ x\ e_2$ implies $e_1 = e_2$
17. lc-unique: If (lc p_1 : lc e) and (lc p_2 : lc e), then lc $p_1 =$ lc p_2

Figure 3: Sample lemmas produced by LNgen

functions over syntax and a large collection of infrastructure lemmas. LNgen automates much of the tedium associated with the locally nameless style, even in our streamlined style, by allowing users to focus on the more interesting aspects of their developments instead of on infrastructure lemmas. In the next section, we describe in additional detail the input to and output from LNgen, highlighting the important properties that are automatically proved.

LNgen is available and has been used for significant developments.[4] LNgen relies on Ott to generate the core locally nameless definitions for a language. It then generates additional definitions and lemmas that are often needed in developmentsthe main benefit that it provides to users over using Ott alone.

The input language for LNgen is a proper subset of the Ott specification language. Figure 2 shows an example input file for untyped lambda terms. The syntax is intended to mimic what

[4]LNgen is available from http://www.cis.upenn.edu/ sweirich/papers/lngen/

one might write informally. Ott is specifically designed for specifying programming languages in a manner that is both convenient for people and machines, e.g., proof assistants. Thus, Ott is a natural starting point for the input language to LNgen. We can take advantage of the work that has gone into the design of Ott, not require users to learn a new specification language, and allow our tool to work in parallel with Ott, relying on Ott for the generation of some of the Coq definitions as well as LATEX output.

Below, we use the example to give a brief overview of the subset of Ott that LNgen supports; a detailed description of the Ott language can be found elsewhere. The first part of an input file for LNgen consists of a list of metavar declarations. Each declaration defines a new type for object language variables LNgen and Ott define binding and substitution for these variables. In Fig. 2, the text repr-locally-nameless indicates that binding should be represented us- ing a locally nameless encoding. (Ott can also output definitions using a concrete representation of binding.) The second part, the grammar, consists of a list of context-free grammar definitions for nonterminals. Each declaration defines a new, inductively defined type for object-language abstract syntax trees. Binding specifications may be attached to each constructor. For example in the `abs` constructor, the metavariable `x` is a binding occurrence in the nonterminal `e`. The third part follows the `substitutions` keyword and indicates that functions for substituting for free variables should be generated. The final part follows the `freevars` keyword and indicates that functions for calculating free variables should be generated. Anything else in the file is ignored by LNgen but may be processed by Ott, e.g., specifications of inductively defined relations.

The main benefit to using LNgen is that it automatically generates a collection of lemmas (with their proofs) about expressions that are useful in metatheoretic reasoning. We highlight the most important of these in Figure 3. The collection shown includes all of the lemmas that we discussed in our previous work. For convenience, LNgen also generates several variants of the lemmas shown and others besides. Our goal in picking the set of lemmas to generate was not to determine some minimal "complete" set for working with metatheory but to generate a set that, from our experience, we know to be useful in formalizations.

LNgen is able to automatically generate the proofs of each of the lemmas in Fig. 3 because, in general, they are "boring" infrastructure lemmas whose proofs are straightforward inductions. At any given point in a proof, there is little choice about what step to take next. Thus, most of the proof scripts start by applying an induction tactic and then use a power tactic to apply a default set of simplifications to the resulting subgoals. In cases where this is not sufficient, LNgen generates more complex scripts based on our knowledge of how such proofs normally proceed. There is no worry about the soundness of our reasoning: the scripts generated by LNgen must be run by Coq to generate proof terms that are then checked. We favor generating proof scripts over proof terms because it keeps the implementation of LNgen simple. Proof terms are specific to individual lemmas and vary from language to language. By contrast, our tacticswhich are useful in their own rightapply to multiple lemmas and do not need to vary from language to language. Unfortunately, because Coqs tactic language is incompletely specified, it is impossible for us to guarantee that our scripts will always succeed. These scripts have never failed on any of our case studies. However, if some proof should fail, the effect is localized. The user may have to do that proof by hand (if they would like to use that lemma) but other generated definitions, lemmas, and proofs will still be available.

Language	OTT LOC	LNgen LOC	Hand LOC
Core-AURA	–	–	12400
STLC	134	1533	108
Contracts [23]	438	3695	3854
Dependent Types [30]	991	7638	12963
Generative Type Abstraction [42]	1544	26197	–

Table 1: Lines of Code Comparison

6 Tool evaluation

To evaluate the genericity of our infrastructure (including LNgen), we used it as part of the three new projects that we were simultaneously working on. These projects are each for general purpose, strongly-typed languages, but their designs have many similarities to AURA. Therefore, we are confident that the lessons that we learned from them will apply to the design of new security-typed languages.

In performing these experiments, we were particularly interested in the following questions: Do generated definitions accurately reflect the language specified by the user? Are lemmas, both those found in our libraries and those that are generated by the tool, useful in practice? Do generated proofs pass Coqs proof checker without modification? In all cases, we found the tool to be correct and useful for our purposes. The representations were accurate, the lemmas important, and we never had any trouble with the output of LNgen.

Table 1 summarize these efforts, listing the lines of code in the specification of the language, the lines of code output by LNgen, and the lines of code written by hand that used the LNgen output. For comparison, we also include the line count from core-AURA, as well as for a small example, the simply-typed lambda calculus (STLC).

Dependent types and program equivalence The definition of type equivalence is one of the most important design issues for any typed language. In dependently-typed languages, because terms appear in types, this definition must rely on a definition of term equivalence. In that case, decidability of type checking requires decidability for the term equivalence relation.

Almost all dependently-typed languages require this relation to be decidable. Some, such as Coq, Epigram or Agda, do so by employing analyses to force all programs to terminate. Conversely, others, such as DML, ATS, Omega, or Haskell, allow nonterminating computation, but do not allow those terms to appear in types. Instead, they identify a terminating index language and use singleton types to connect indices to computation. In both cases, decidable type checking comes at a cost, in terms of complexity and expressiveness.

Conversely, the benefits to be gained by decidable type checking are modest. Termination analyses allow dependently typed programs to verify total correctness properties. However, decidable type checking is not a prerequisite for type safety. Furthermore, decidability does not imply tractability. A decidable approximation of program equivalence may not be useful in practice.

The paper "Dependent types and Program Equivalence" [30], takes a different approach: instead of a fixed notion for term equi valence, we parameterize our type system with an abstract relation that is not n necessarily decidable. We then design a novel set of typing rules that require only weak properties of this abstract relation in the proof of the preservation and progress lemmas. This design provides flexibility: we compare valid instantiations of term equivalence which range from beta-equivalence, to contextual equivalence, to some exotic equivalences.

Because the goal of this work was to make the exact properties of program equivalence that are required for type soundness precise, it made sense to formalize this work using the Coq proof assistent. This project was an excellent case study for LNgen: the specification of the entire language fell within the capabilities of the LNgen tool. Therefore, we used Ott to specify the language and LNgen to generate its output. Once this was set up, it took less than two weeks to state and prove the desired properties of the language—the experience of AURA as well as the availability of the LNgen tool meant that the task of producing a machine-checked version of the programming language metatheory was not a significant cost to the project. Furthermore, confidence in our results was significantly increased due to the presence of these proofs. LNgen was an important part of the project—without the availability of LNgen, we would not have been able to complete the proofs before the conference deadline. Because the tool provided every infrastructure lemma we needed, we were able to focus their efforts on the novel aspects of their languages design.

Contracts made Manifest Higher-order contracts, introduced by Findler and Felleisen, are a language construct that extend dynamic contract checking to languages with first-class functions. Since their introduction, many variants have been proposed. Broadly, these fall into two groups: some follow Findler and Felleisen in using latent contracts, purely dynamic checks that are transparent to the type system; others use manifest contracts, where refinement types record the most recent check that has been applied to each value. These two approaches are commonly assumed to be equivalent–different ways of implementing the same idea, one retaining a simple type system, and the other providing more static information.

The paper "Contracts Made Manifest" [23] extends the work of Gronski and Flanagan, who defined a latent calculus λ_c and a manifest calculus λ_h, gave a translation ϕ from λ_c to λ_h, and proved that, if a λ_c term reduces to a constant, then so does its ϕ image. We enrich their account with a translation ψ from λ_h to λ_c and prove an analogous theorem. The paper then generalizes the whole framework to dependent contracts, whose predicates can mention free variables. This extension is both pragmatically crucial, supporting a much more interesting range of contracts, and theoretically challenging. We define dependent versions of λ_h and two dialects (lax and picky) of λ_c, establish type soundness—a substantial result in itself, for λ_h—and extend ϕ and ψ accordingly. Surprisingly, the intuition that the latent and manifest systems are equivalent now breaks down: the extended translations preserve behavior in one direction but, in the other, sometimes yield terms that blame more.

In the process of this work, we found that there was one part that was particularly well-suited to formalization. Most of the work went beyond the capabilities of tools like LNgen (requiring type-directed translations and the definitions of logical relations). However, one particular thorny part of the work relied on properties of reduction of the term language. This part was intricate, but not very enlightening to the project as a whole. The sixty page technical report that accompanies this paper and provides (paper) proofs of the theorems, included this part as an appendix. Therefore, we decided to mechanically verify the results. Graduate student Michael Greenberg, who did not have prior experience with AURA, was nevertheless able to use the output of LNgen to prove the necessary results in Coq in about one month worth of effort. Greenberg reported that "All in all, LNgen was great—it covered most of the stupid facts I needed." The tool failed to generate only one set of lemmas, which concerned how substitution maintains invariants about the free variables of terms.

Generative Type Abstraction and Type-level Computation Modular languages support generative type abstraction, ensuring that an abstract type is distinct from its representation, except inside the implementation where the two are synonymous. This well-established feature is in tension with the non-parametric features of newer type systems, such as indexed type families and GADTs. In the paper "Generative Type Abstraction and Type-level Computation" [42] we solve the problem by using kinds to distinguish between parametric and non-parametric contexts. The result is directly applicable to the Haskell programming language, which is rapidly developing support for type-level computation, but the same issues should arise whenever generativity and non-parametric features are combined.

We involved the Ott and the LNgen tool early in the development of this work, using these tools to specify the language under study, ensure that we had a complete specification, and verify consistency properties of that specification. As Figure 1 demonstrates, the language studied by this paper is the largest case study that we have performed. Because of the size of the language and the nature of the project we never mechanically proved any results about that specification—our reasoning was on paper only. Although the LNgen specification evolved as the design evolved, we ultimately abandoned the LNgen part of the specification. (The line counts are from the last version that worked with LNgen.) One reason for this decision was time pressure—we were working to a paper deadline, and this part was not on the critical path. Another reason was that we choose to include features in the language that went beyond the capabilities of LNgen.

However, even as a pure specificational tool during the design process, making the system go through Ott and LNgen to produce a Coq specification was valuable. The fact that we could type check the Coq output means that certain bugs in the spec were eliminated much earlier than they would have been. Furthermore, the binding infrastructure produced by LNgen was beneficial to the small experiments that we did on the specification, even if we never produced a complete mechanical proof of our results.

7 Conclusions

Security-oriented languages are a key ingredient to the design of secure software systems. The provide specific functionality, such as mechanisms for authorization, auditing, confidentiality, and the tracking of information-flow throughout the system. Programs written in these languages that take advantage of these features, can be automatically shown to possesses key properties, essential to the security of the entire system. However, these results critically rely on the metatheoretic properties of the languages themselves. This project has demonstrated both that it is feasible to mechanically check these properties, and that the overheads of the engineering process of producing mechanically verified languages are reasonable.

References

[1] Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon Riecke. A core calculus of dependency. In *Proc. 26th ACM Symp. on Principles of Programming Languages (POPL)*, pages 147–160, San Antonio, TX, January 1999.

[2] Amal J. Ahmed, Andrew W. Appel, and Roberto Virga. A stratified semantics of general references embeddable in higher-order logic. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science (LICS 2002)*, pages 75–86, Copenhagen, Denmark, July 2002.

[3] Andrew W. Appel. Foundational proof-carrying code. In *16th Annual IEEE Symposium on Logic in Computer Science (LICS '01)*, June 2001.

[4] Brian Aydemir, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich. Engineering formal metatheory. In *ACMSIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 3–15, January 2008.

[5] Brian Aydemir and Stephanie Weirich. Lngen: Tool support for locally nameless representations. Technical Report MS-CIS-10-24, Computer and Information Science, University of Pennsylvania, June 2010.

[6] Anindya Banerjee and David A. Naumann. Secure information flow and pointer confinement in a Java-like language. In *Proc. of the 15th IEEE Computer Security Foundations Workshop*, 2002.

[7] G. Barthe, D. Pichardie, and T. Rezk. A certified lightweight non-interference Java bytecode verifier. Technical report, INRIA, 2006. `http://hal.inria.fr/inria-00106182`.

[8] Gilles Barthe, David Pichardie, and Tamara Rezk. A certified lightweight non-interference Java bytecode verifier. In Rocco De Nicola, editor, *ESOP*, volume 4421 of *Lecture Notes in Computer Science*, pages 125–140. Springer, 2007.

[9] Elisa Bertino, Barbara Catania, Elena Ferrari, and Paolo Perlasca. A logical framework for reasoning about access control models. *ACM Trans. Inf. Syst. Secur.*, 6(1):71–127, 2003.

[10] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development*, volume XXV of *EATCS Texts in Theoretical Computer Science*. Springer-Verlag, 2004.

[11] Steven Bishop, Matthew Fairbairn, Michael Norrish, Peter Sewell, Michael Smith, and Keith Wansbrough. Rigorous specification and conformance testing techniques for network protocols, as applied to TCP, UDP, and Sockets. In *Proceedings of SIGCOMM 2005 (Philadelphia)*, August 2005.

[12] Matt Blaze, Joan Feigenbaum, and Jack Lacy. Decentralized trust management. Technical Report 96-17, 28, 1996.

[13] Juan Chen and David Tarditi. A simple typed intermediate language for object-oriented languages. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 38–49, New York, NY, USA, 2005. ACM Press.

[14] Adam Chlipala. A certified type-preserving compiler from lambda calculus to assembly language. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI'07)*, 2007.

[15] Karl Crary. Toward a foundational typed assembly language. In *Symposium on Principles of Programming Languages*, January 2003.

[16] Dorothy E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, May 1976.

[17] Dorothy E. Denning and Peter J. Denning. Certification of Programs for Secure Information Flow. *Comm. of the ACM*, 20(7):504–513, July 1977.

[18] John DeTreville. Binder, a logic-based security language. In M.Abadi and S.Bellovin, editors, *Proceedings of the 2002 Symposium on Security and Privacy (S&P'02)*, pages 105–113, Berkeley, California, May 2002. IEEE Computer Society Press.

[19] Office of Secretary of Defense Director, Force Transformation. The implementation of network-centric warfare, January 2005. `http://www.oft.osd.mil`.

[20] R. J. Feiertag, K. N. Levitt, and L. Robinson. Proving multilevel security of a system design. *Proc. 6th ACM Symp. on Operating System Principles (SOSP), ACM Operating Systems Review*, 11(5):57–66, November 1977.

[21] Cédric Fournet, Andrew D. Gordon, and Sergio Maffeis. A type discipline for authorization policies. In *14th European Symposium on Programming, ESOP 2005*, volume 3444 of *Lecture Notes in Computer Science*, pages 141–156. Springer-Verlag, 2005.

[22] J. A. Goguen and J. Meseguer. Security policies and security models. In *Proc. IEEE Symposium on Security and Privacy*, pages 11–20. IEEE Computer Society Press, April 1982.

[23] Michael Greenberg, Benjamin Pierce, and Stephanie Weirich. Contracts made manifest. In *37th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 353–364, Madrid, Spain, January 2010. ACM.

[24] J. Halpern and V. Weissman. Using first-order logic to reason about policies. In *Proceedings of the 16th IEEE Computer Security Foundations Workshop (CSFW'03)*, 2003.

[25] Robert Harper and Chris Stone. A type-theoretic interpretation of Standard ML. In Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language, and Interaction: Essays in Honor of Robin Milner*. MIT Press, 2000.

[26] Galen Hunt and James Larus. Singularity: Rethinking the Software Stack. *Operating Systems Review*, 41(2):37–49, April 2007.

[27] B. Jacobs, W. Pieters, and M. Warnier. Statically checking confidentiality via dynamic labels. In *Workshop on Issues in the Theory of Security (WITS'05)*, 2005.

[28] Limin Jia, Jeffrey A. Vaughan, Karl Mazurak, Jianzhou Zhao, Luke Zarko, Joseph Schorr, and Steve Zdancewic. Aura: A programming language for authorization and audit. In *Proc. of the 13th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, Victoria, British Columbia, Canada, September 2008.

[29] Limin Jia and Steve Zdancewic. Encoding information flow in aura. In *Workshop on Programming Languages and Analysis for Security (PLAS)*, pages 17–29, 2009.

[30] Limin Jia, Jianzhou Zhao, Vilhem Sjöberg, and Stephanie Weirich. Dependent types and program equivalence. In *37th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 275–286, Madrid, Spain, January 2010. ACM.

[31] Trevor Jim. SD3: a trust management system with certificate revocation. In *IEEE Symposium on Security and Privacy*, pages 106–115, 2001.

[32] Matt Kaufmann, J Strother Moore, and Panagiotis Manolios. *Computer-Aided Reasoning: An Approach.* Kluwer Academic Publishers, 2000.

[33] Gerwin Klein and Tobias Nipkow. A machine-checked model for a Java-like language, virtual machine and compiler. Technical Report 0400001T.1, National ICT Australia, Sydney, March 2004.

[34] Daniel K. Lee, Karl Crary, and Robert Harper. Towards a mechanized metatheory of Standard ML. In *POPL '07: Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 173–184. ACM Press, 2007.

[35] Xavier Leroy. Formal certification of a compiler back-end or: Programming a compiler with a proof assistant. In *POPL '06: Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 42–54. ACM Press, 2006.

[36] Ninghui Li, John C. Mitchell, and William H. Winsborough. Beyond proof-of-compliance: security analysis in trust management. *J. ACM*, 52(3):474–514, 2005.

[37] Neophytos G. Michael and Andrew W. Appel. Machine instruction syntax and semantics in higher order logic. In *17th International Conference on Automated Deduction (CADE-17)*. Springer-Verlag (Lecture Notes in Artificial Intelligence), June 2000.

[38] Andrew C. Myers. JFlow: Practical mostly-static information flow control. In *Proc. 26th ACM Symp. on Principles of Programming Languages (POPL)*, pages 228–241, San Antonio, TX, January 1999.

[39] Andrew C. Myers, Stephen Chong, Nathaniel Nystrom, Lantian Zheng, and Steve Zdancewic. Jif: Java information flow. Software release. Located at http://www.cs.cornell.edu/jif, July 2001.

[40] David Naumann. Verifying a secure information flow analyzer. In *Theorem Proving in Higher Order Logics (TPHOLs)*, 2005.

[41] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant For Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science.* Springer-Verlag, 2002.

[42] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Steve Zdancewic. Generative type abstraction and type-level computation. To appear in POPL 2011, 2011.

[43] Frank Pfenning and Carsten Schürmann. System description: Twelf — A meta-logical framework for deductive systems. In Harald Ganzinger, editor, *Automated Deduction, CADE 16: 16th International Conference on Automated Deduction*, volume 1632 of *Lecture Notes in Artificial Intelligence*, pages 202–206. Springer-Verlag, 1999.

[44] François Pottier and Vincent Simonet. Information flow inference for ML. In *Proc. 29th ACM Symp. on Principles of Programming Languages (POPL)*, Portland, Oregon, January 2002.

[45] Alejandro Russo, Koen Claessen, and John Hughes. A library for light-weight information-flow security in haskell. In *Haskell '08: Proceedings of the first ACM SIGPLAN symposium on Haskell*, pages 13–24, New York, NY, USA, 2008. ACM.

[46] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, January 2003.

[47] Fred B. Schneider, editor. *Trust in Cyberspace*. National Academy Press, 1999.

[48] Fred B. Schneider, Greg Morrisett, and Robert Harper. A language-based approach to security. In Reinhard Wilhelm, editor, *Informatics: 10 Years Back, 10 Years Ahead*, volume 2000 of *Lecture Notes in Computer Science*, pages 86–101. Springer-Verlag, Heidelberg, 2001.

[49] Peter Sewell, Francsco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strniša. Ott: Effective tool support for the working semanticist. In *(ICFP '07) Proc. of the 2007 ACM SIGPLAN International Conf. on Functional Programming.*, pages 1–12. ACM, 2007.

[50] Rui Shi. Implementing reliable Linux device drivers in ATS. In *PLPV '07: Proceedings of the 2007 Workshop on Programming Languages Meets Program Verification*, pages 41–46. ACM, 2007.

[51] Martin Strecker. Formal analysis of an information flow type system for MicroJava (extended version). Technical report, Technische Universität München, July 2003.

[52] Stephen Tse and Steve Zdancewic. Designing a Security-typed Language with Certificate-based Declassification. In *Proc. of the 14th European Symposium on Programming*, 2005.

[53] US national commission on terrorist attacks upon the United States, 9/11 commission report, 2004. Available at `http://www.9-11commission.gov`.

[54] Jeffrey A. Vaughan. *AURA: Programming with Authorization and Audit*. PhD thesis, University of Pennsylvania, 2009.

[55] Jeffrey A. Vaughan, Limin Jia, Karl Mazurak, and Steve Zdancewic. Evidence-based audit. In *Proc. of 21st IEEE Computer Security Foundations Symposium (CSF)*, pages 177–191. IEEE Computer Society Press, 2008.

[56] Dennis Volpano, Geoffrey Smith, and Cynthia Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1996.

[57] Feng Zhou, Jeremy Condit, Zachary R. Anderson, Ilya Bagrak, Robert Ennals, Matthew Harren, George C. Necula, and Eric A. Brewer. Safedrive: Safe and recoverable extensions using language-based techniques. In *7th Symposium on Operating Systems Design and Implementation (OSDI '06), November 6-8, Seattle, WA, USA*, pages 45–60. USENIX Association, 2006.